

Practicum in Database Systems

Project 3 intro



Project 3 Introduction

- Add B+ tree indexing support to your SQL interpreter
 - build indexes
 - use indexes for selection



Project 3 Introduction

- Somewhat independent of P2
 - Same binary I/O format for data
 - Will be testing your queries only with TNLJ and in-memory sort implementations
 - But will need the others for P4, so you're not off the hook for those



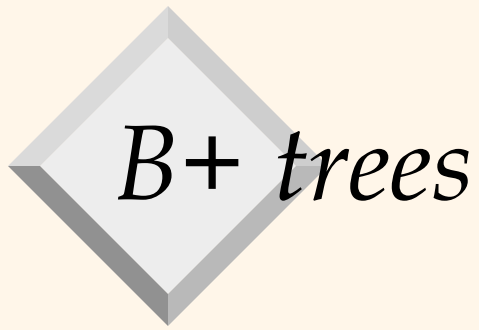
Scope and length

- *Should* be significantly shorter than P2
- But don't take it lightly
 - since you have less time




Your TODOs

- Implement a bulk loading algorithm for B+-tree construction
 - And to serialize B+-tree to a file
- Implement an Index Scan operator that uses your tree index to retrieve records
- Integrate your index scan with your interpreter so it can be used for selection



B+ trees

- You are familiar with them from 4320
- But you know some details are implementation-specific
- Be sure to read the 4321 instructions, Section 2, very carefully!



What goes in the leaves

- Alternative 3
 - Every data entry is a key with a list of record ids
 - A record id is a (page id, tuple id) pair
- That way you won't need to worry about duplicate keys in the index
- But data entries are variable-length



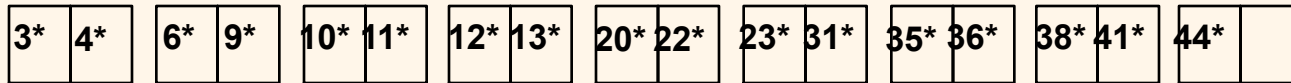
Bulk loading

- Good news – no insert/delete needs to be implemented as your data is static
- But need to build the tree via **bulk loading**
- A number of different algorithms
 - Your textbook has one – don't use that one
 - The one you should use is in the instructions



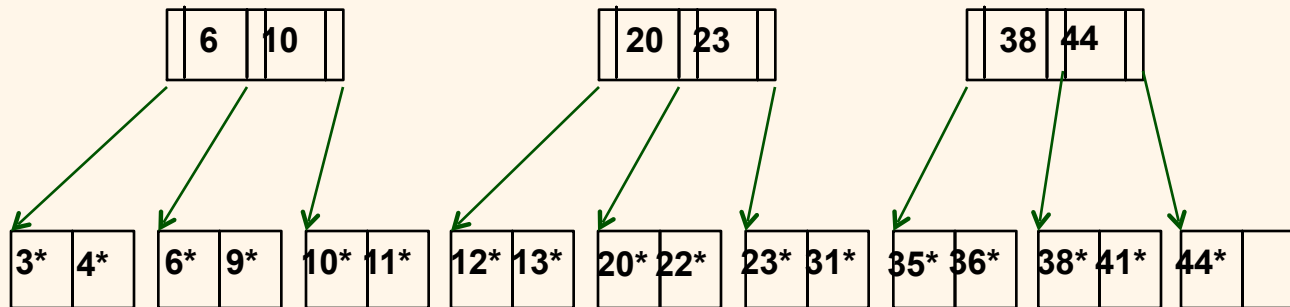
Bulk Loading of a B+ Tree

- *Initialization:*
- Create and sort all data entries
- Create all the leaf nodes



Bulk Loading of a B+ Tree

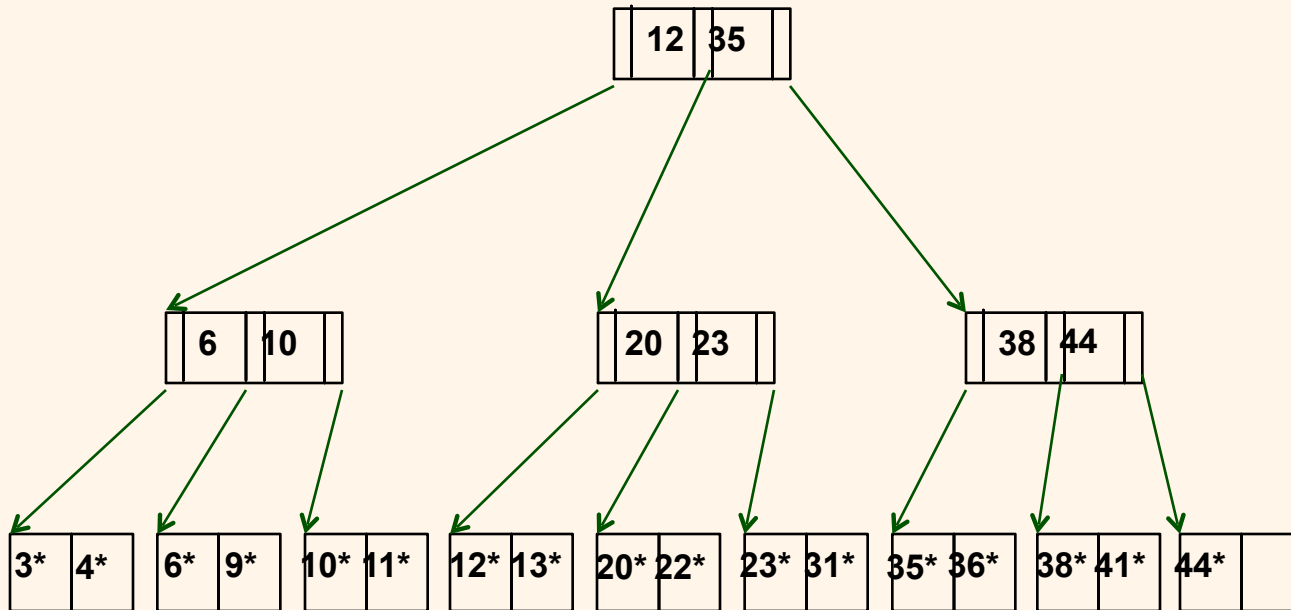
- Now create next layer above the leaf nodes
- Can control how full you want them to leave space for future inserts
 - In this project, we want them as full as possible

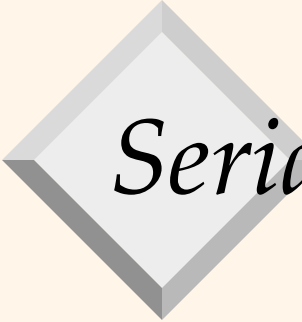




Bulk Loading of a B+ Tree


- Now build next layer above that
- Repeat until obtain a one-node layer – that's your root





Serializing a B+-tree to a file

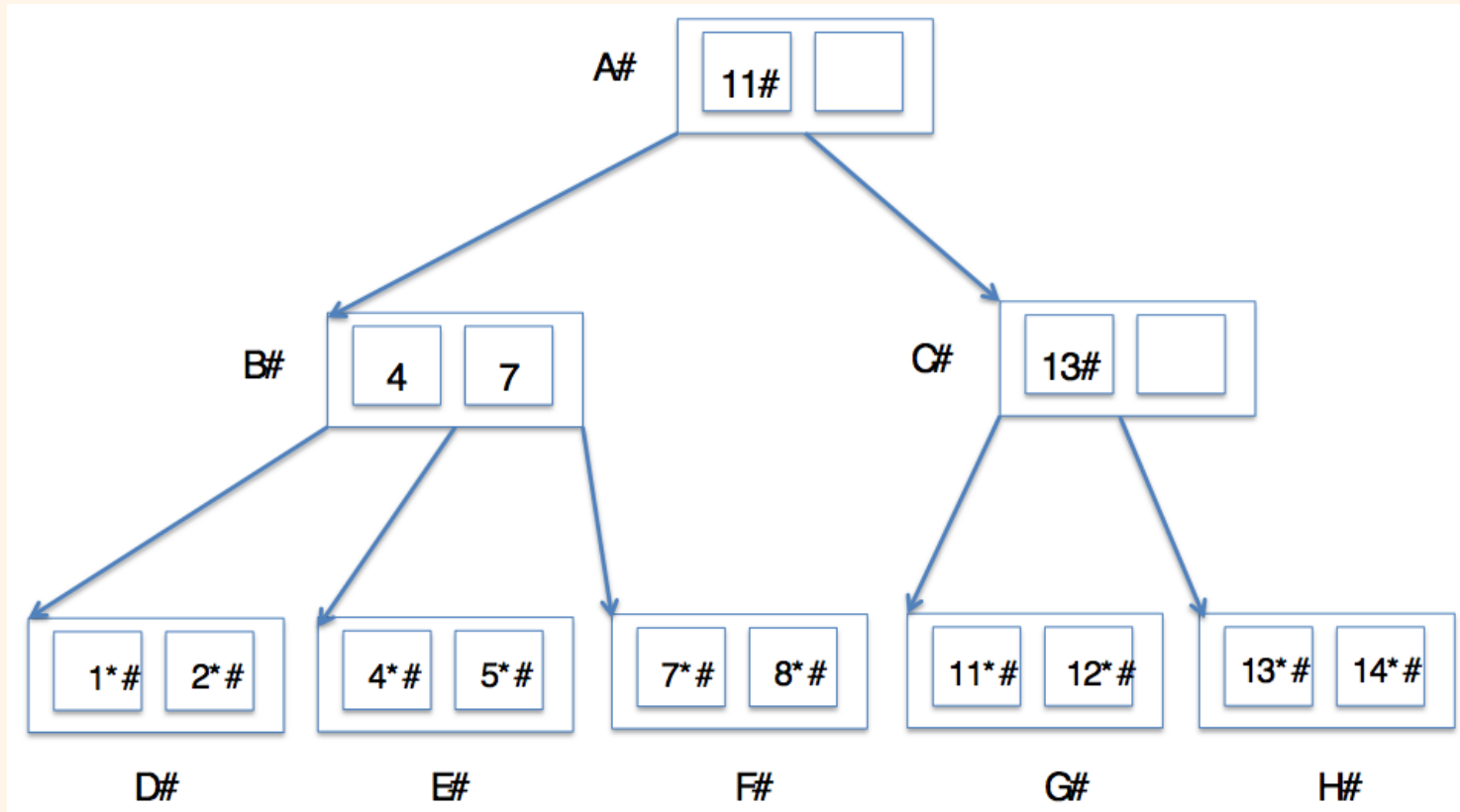
- Indexes don't sit in memory all the time
- So we need a binary format to serialize them
- One node = one page
 - Whether index node or leaf node
 - May assume every node will fit on a 4096-byte page when serialized in our format



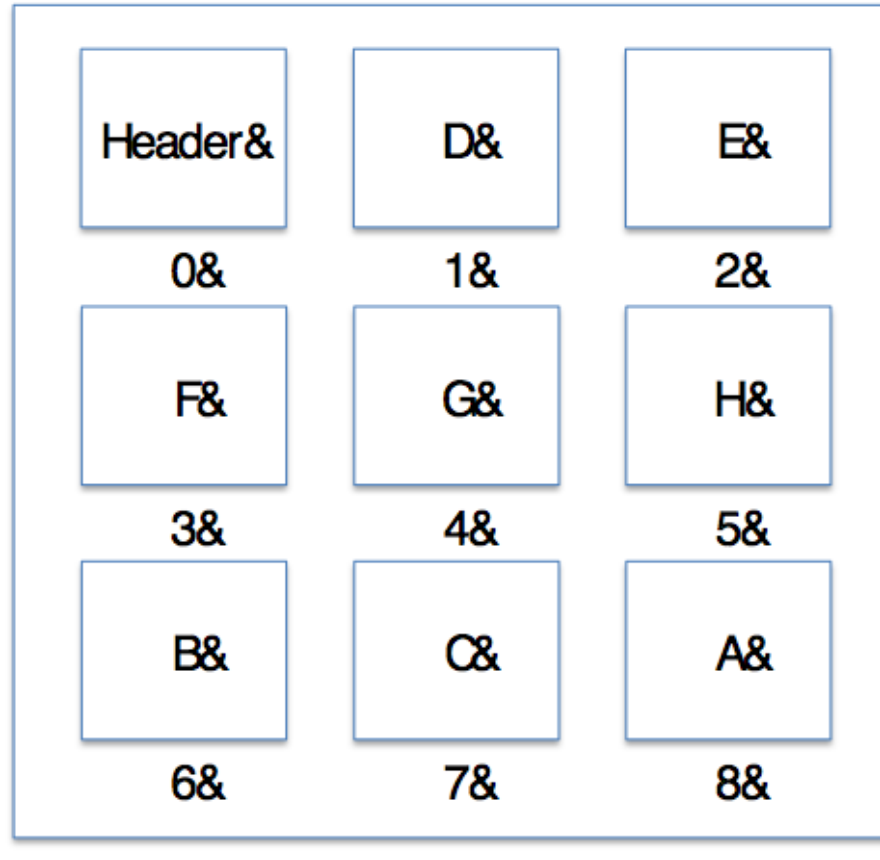
Overall file format for a tree

- Header page
- Leaf pages in order left-to-right
- Index pages from layer immediately above leaves
- Etc
- Root is on last page

Example tree



Serialization in file





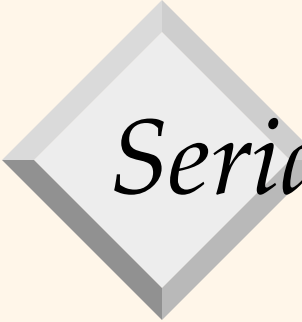
Layout and addresses

- Leaf nodes sequential in file, so no need for next/prev pointers in leaves
- A node's *address* is the page it is stored on
 - Not known until the node is serialized
 - Index nodes store the addresses of their children
 - Suggests why the leaf nodes are serialized first in the file (serialize one layer at a time, setting addresses in parent layer as you go)



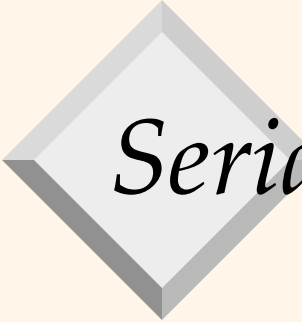
Header page

- Basic info about the tree:
 - Address of the root
 - Number of leaves
 - Order (parameter d)
- In a "real" system this might store more info such as the type of the key (string, integer etc)



Serializing a node to a page

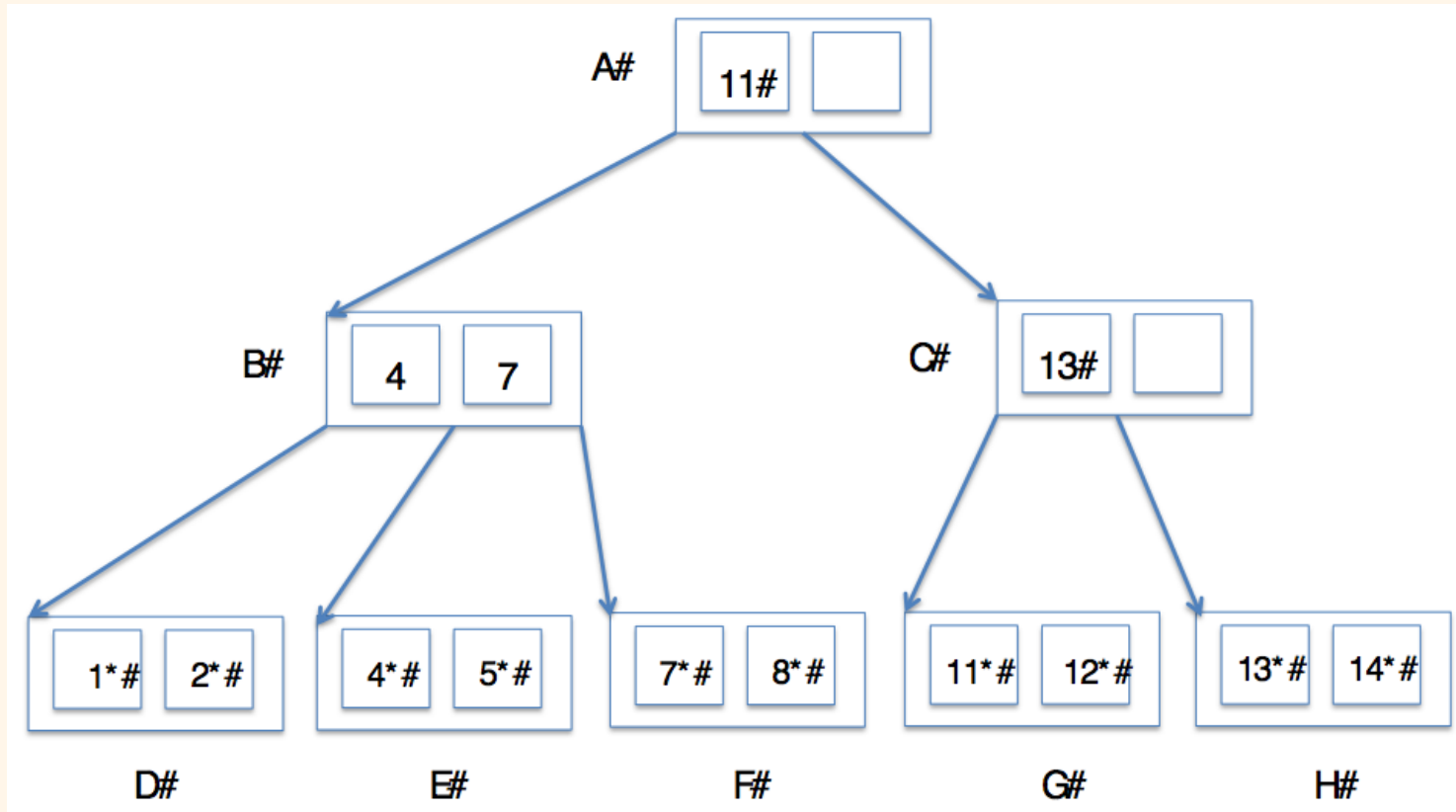
- Format depends on whether node is an index node or a leaf node



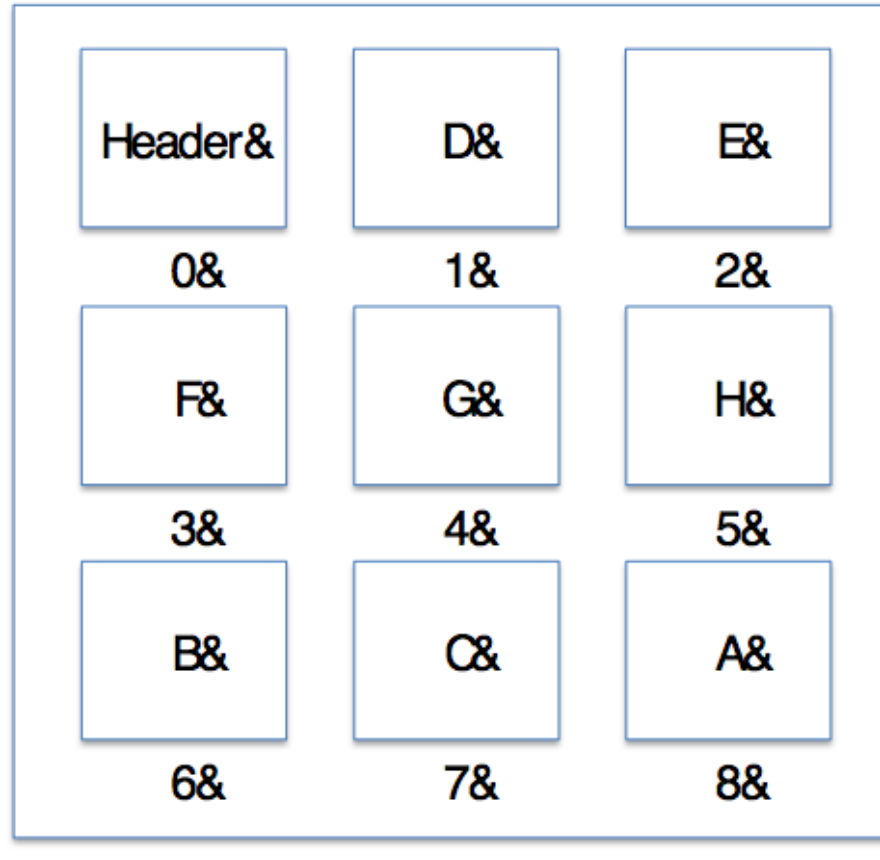
Serializing an index node

- Flag 1 to indicate it's an index node
- Number of keys
- Keys
- Addresses of children

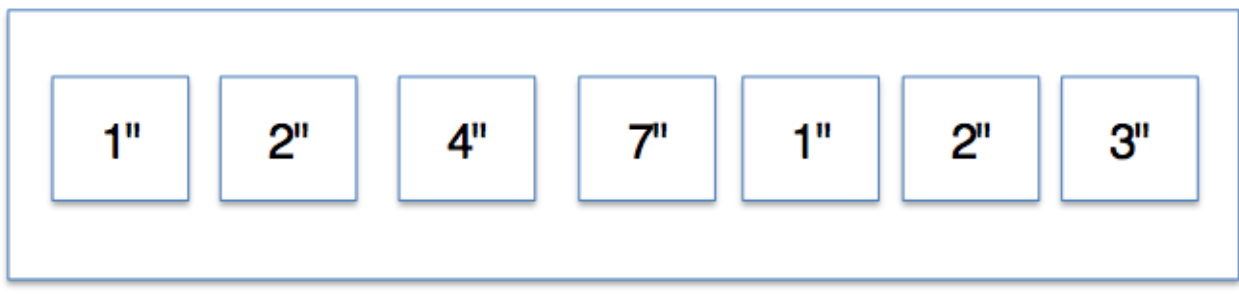
Example tree

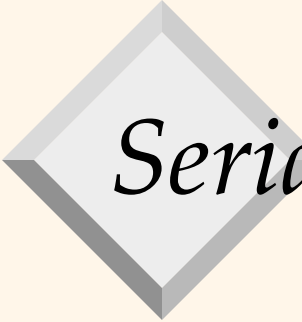


Serialization in file



Serializing node B

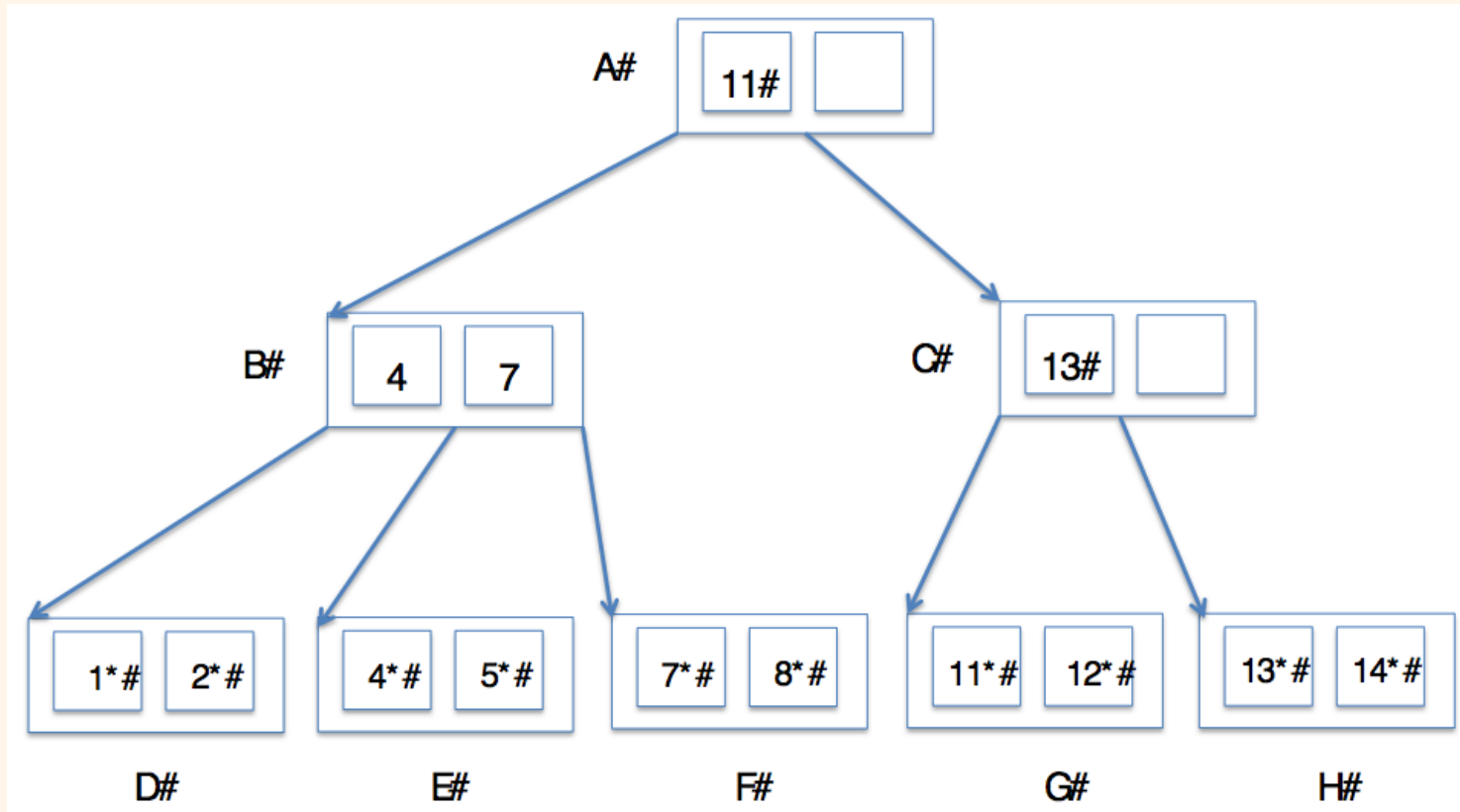




Serializing a leaf node

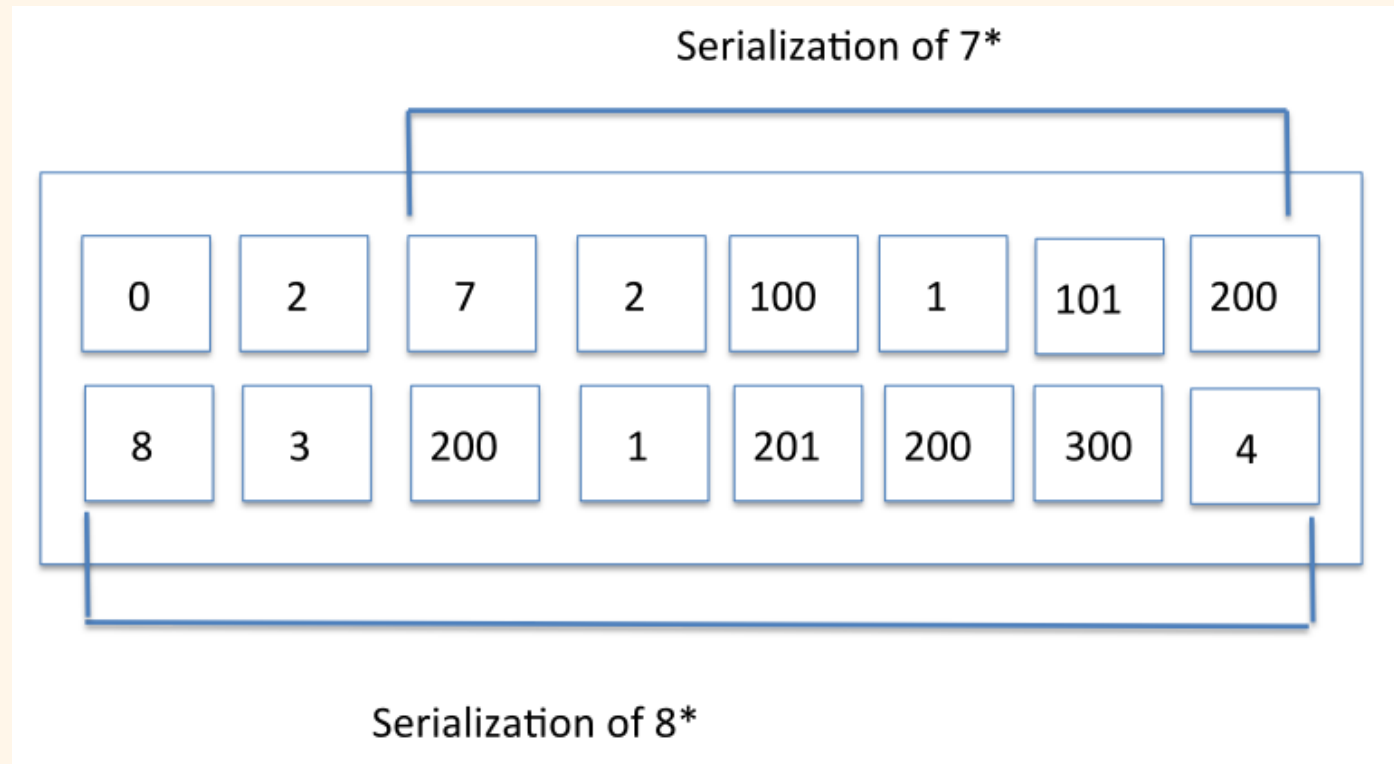
- Flag 0 to indicate it's a leaf node
- Number of data entries
- A serialized representation of each data entry


Example tree



Serializing node F


- Suppose 7^* is $\langle 7, [(100, 1), (101, 200)] \rangle$
- Suppose 8^* is $\langle 8, [(200,1), (201, 200), (300,4)] \rangle$






Where do your indexes go?

- Some new files and folders in your directory structure
- *index_info.txt* specifies what indexes to build
- *indexes* subdirectory contains the serialized indexes
- At most one index per relation
- Only build indexes on a single attribute



Configurations

- We want to run your code in three different ways:
 - Build indexes (only)
 - Run queries using provided indexes (our indexes)
 - `index_info` file provides guidance to the DB catalog as to what indexes are available for use in selection
 - Build indexes and run queries



Yet another config file...

- Your interpreter now takes a config file instead of command-line arguments
- First 3 lines specify the 3 arguments from P2 (input/output/temporary sort)
- Next 2 lines are flags for whether to build indexes and whether to run queries



Plan builder config file

- Previously, this specified the join and sort implementations
- Now it also specifies whether or not to try using an index for selection



Your first task

- Implement an algorithm that builds a tree index and serializes it to a file
- If we desire a clustered index, should start by sorting the relation file by the indexed attribute
- Should be relatively straightforward once you understand the format
- Connect your algorithm to your interpreter so you can build all the desired indexes from `index_info.txt`




Your second task

- Now, time to use your indexes for selection
- Implement a new physical operator:
IndexScan
 - You had a scan previously that read the whole file
 - Now you're going to retrieve tuples using the index instead!



IndexScan

- Every IndexScan has a lowKey and a highKey
- These specify the scan range you want
 - e.g from 1 to 2000
 - one (or theoretically both) may be null, to indicate you want to scan from the beginning or to the end of the key space



What does the IndexScan do?

- On initialization, grab the index file and deserialize nodes from the root to the first relevant leaf based on lowKey
 - Do NOT deserialize the whole tree!!!
- Upon calls to getNextTuple(), obtain the next tuple using the appropriate info from your index
 - until you run out of tuples or reach highKey



Implementing IndexScan

- If your index is *clustered*, don't read any leaves after the first one
 - Jump into the data file which is sorted and read tuples from there
- If your index is *unclustered*, you'll have to scan the leaves and resolve the rid to a particular tuple in the data file
 - I push the implementation of this to my TupleReader, so it can now effectively do "random access" into the file based on rid



Using IndexScan for selection

- Your third task: make your PhysicalPlanBuilder use indexes for selection
- Assume you are pushing selections so your logical selection operators all have leaves as children
- A general selection may need to be implemented as two operators:
 - an IndexScan to handle whatever portion of the selection can be dealt with using index
 - followed by a "regular" full-scan selection for the remainder



What your PlanBuilder needs to do

- Check whether there is an index on the relation in question and whether it's clustered
- Partition the selection condition into the part that can be handled via the index and the part that cannot
 - E.g. $R.A < 5 \text{ AND } R.C \geq 10 \text{ AND } R.B = 3$, index on R.B...
 - Visitor pattern !!




What your PlanBuilder needs to do

- Translate the selection condition into:
 - A lowkey/highkey pair for your IndexScan
 - The "remainder" that cannot be handled via the index
 - One of those may be null
- Create appropriate physical operators



Finally

- Like in P2, we want you to do some performance benchmarking to see if indexing helps your queries run faster



Must-have requirements

- Implement IndexScan following our logic/
instructions
 - lowKey and highKey
 - don't deserialize entire tree
 - handle clustered/unclustered indexes differently
- Use IndexScans to implement selection
 - handle maximal possible portion of selection via index
- As in P2, give a good-faith implementation of everything we request or tell us in the README if you don't